Department of Computing and Information Systems

COMP30023: Computer Systems

Project 2

Due date: No later than 5:00pm on Friday, May 27. **Weight:** 15%

Overview

The aims of Project 2 are:

- To provide you with experience in writing programs that interact with each other over a network (socket programming).
- To provide you with experience in writing multi-threaded applications and process synchronization.

Your task consists of three components:

- To implement a `server` capable of handling multiple concurrent requests for the game 'Mastermind'.

  See `https://en.wikipedia.org/wiki/Mastermind_(board_game)` for an overview of the game.
- To implement a `client` program that can be used to test your game `server`.
- To report on `server` resources usage (performance metrics and other interesting statistics).

Note: a 'simple' game was selected for the project, so that you can devote all (most) of your effort to the concurrency issues rather than getting 'bogged down' with game playing strategies.

Your programs must be written in C.

The executables programs must be called `server` and `client` respectively. Your programs must run on both the School of Engineering Linux server `digitalis.eng.unimelb.edu.au` and your NeCTAR cloud VM (your executable must run on the OS configuration adopted, with an accessible port).

Rules of Mastermind (for this project)

Mastermind is a two-player game consisting of a 'codemaker' and a 'codebreaker.' The codemaker secretly selects a code consisting of an ordered sequence of four colours $c_1c_2c_3c_4$, each chosen from the set {`A, B, C, D, E, F`} of six possible colours, with repetitions allowed.

The codebreaker then tries to guess the code by repeatedly submitting their nominated sequence of four colours from the set {`A, B, C, D, E, F`}.

After each guess, the codemaker provides feedback to the codebreaker using two numbers: the number of correct colours in the correct positions `b` and the number of colours that are part of the code but not in the correct positions `m`, using the format `[b:m]`. For example, if the code is `ABCC` and the codebreaker's guess is `BCDC`, then the codemaker's response would be `[1:2]` since the codebreaker has guessed the second `C` correctly and in the correct position, while having guessed the `B` and the first `C` correctly, but in the wrong position.

The codebreaker continues guessing until he/she guesses the code correctly, or until he/she reaches a maximum allowable number of guesses (set to 10 in this project) without having correctly identified the secret code.

Requirements

We are not imposing tight constraints on design/requirements – you are free to design your `server` and `client` programs as you see fit. However, your implementation must meet the criteria described below:

Implementation overview

- Your `server` program will play the role of the 'codemaker.' Your `client` program will play the role of the 'codebreaker.'
- Your `server` program must be able to handle multiple/concurrent requests from individual `clients` (typically running on hosts with different IP addresses to the server). See Figure 1.
  - that is, your `server` program must be able to carry on multiple game sessions with separate clients simultaneously, rather than delaying new connections until the current session is finished
- You must make use of `Pthreads` to process each `client`'s interactions with the `server`.
- When you implement sockets, you must use TCP (SOCK_STREAM for AF_INET).
- Your `server` program uses the following command line arguments:
  - a port number
  - a default four character string representing the secret code (e.g., `AAAA`) that can be used for testing your code — if this argument is not supplied, your server program will randomly generate the secret code
- Your `client` program uses the following command line arguments:
  - an IP address (or host name) of the `server`
  - the corresponding port number
- It is important to note that `clients` do not play games against each other. ie. a `client` plays a game against the `server`.

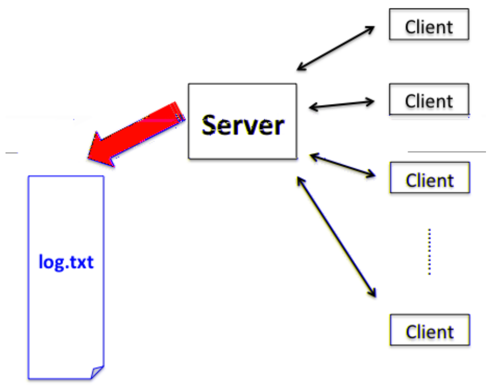

Figure 1: *A simple illustration of the client-server architecture. Your server must be able to handle concurrent requests. The server is responsible for maintaining a log file*(`log.txt`) *of interactions between each client and the server.*

Playing the game

- Your `server` program should send an appropriate 'Welcome' message to the `client` program, explaining the rules of the game (the format of the message is up to you!)
- The client-server interaction general works as follows:
  - the `server` creates the secret code
  - the `server` requests the `client` to submit a guess of the secret code
  - the `client` 'sends' their guess to the `server`
  - the `server` checks the guess and provides feedback
  - this cycle continues until either *(a)* the `client` successfully guesses the secret code, or *(b)* the `client` uses up the maximum number of guesses allowed (10 in this case).
- Your `client` program should display text-based message describing the status of the game (with appropriate messages) for each step of the game, so that the 'human' player has the necessary information to make a guess of the secret code.
- To keep things simple, the only input you collect from the 'human' player (client) on each turn is a string of four `char` values representing their guess. For example, the 'human player' (client) enters `ABCD` representing `A` in column 1, `B` in column 2, `C` in column 3 and `D` in column 4. The `client` then transfer this input string (or nominated guess) to the `server` for processing.
- You do not have to do error checking of user input (on the `client` side) – assume that the user enters a four character string (although, they could possibly choose characters outside of the expected range {`A, B, C, D, E, F`}).
- Based on the input submitted by `client` program, your `server` program should send an appropriate response:
  - the feedback values `[b:m]` indicating the number of matching parts of the code (e.g. `[1:2]` based on the example in the Rules section above), or
  - an `INVALID` message indicating that the guess submitted was not valid (e.g., the user entered `ZZAA`), or
  - a `SUCCESS` message indicating that the secret code has been identified

    Note: the `client` should close the connection on receipt of the `SUCCESS` message.
- Within a game session, the `server` program should keep track of how many guesses the `client` program has made. If the `client` program has not identified the secret code after 10 attempts, the `server` program sends
  - a `FAILURE` message indicating that there are 'no more attempts' as well as displaying the secret code

    Note: the `client` should close the connection on receipt of the FAILURE message.

- You must continually write to a log file (`log.txt`) detailing interactions between the `server` and individual `clients`. Each log file entry will include:
  - a time-stamp (system date/time)
  - IP address of the `client` or `0.0.0.0` for the `server`
  - a socket id (or file descriptor) for the `client` connection
  - details of the exchange between client and server.
    * from the `server` – record the secret key
    * from the `client` – record the guess (e.g. `BCDC`) submitted
    * from the `server` – record an appropriate message (e.g, `[1:2]` or `INVALID` or `SUCCESS` or `FAILURE` with a 'game over' message)

  See the LMS for an example log file. Note: we will not use the `diff` command to check your output file against an expected output.

- Care must be taken to make sure that the concurrent processing of interactions between the `server` and individual `clients` does not cause incorrect entries in the log file.

Investigating `server` resource usage / performance

In this subtask, you are asked to examine overall `server` 'performance' and resource usage and write the results to the bottom of the log file (`log.txt`) when your `server` program shuts down – you write to the log file after the key combination `Ctr+C` is used to 'terminate' your process.

- In the simplest case, you could report the number of `clients` that successfully connected to your `server` (and a count of the number of clients who successfully guessed the secret code).
- You should also investigate resource usage (e.g., `rusage`, including `timeval`, and/or memory use in Pthreads etc).
- It may also be interesting to examine the `/proc` (virtual file system) and other data the you can can access via the *process id*.

  Note: we are not being prescriptive here – your task is to identify and report on what you consider to be 'interesting' statistics. If you are measuring memory usage for Pthreads, you may need to consider using a `mutex` when summing/accumulating memory usage across individual threads.

Typically, the key combination `Ctr+C` is used to terminate processes in Linux. To shut down your `server` program we will use this key combination. However, you should write an appropriate signal handler function to 'catch' the `Ctr+C` input, which will:

- write the 'performance' data (statistics) to the bottom of the log file (`log.txt`).
- then, terminate your `server` program.

Report – describing resource usage / performance

Prepare a short report ($\leq$ 500 words in length) named `report.txt` using plain text format that:

- describes the **resource usage and performance data** that you have collected.
- compares the values for resource usage and performance data when your `sever` program runs on both `digitalis.eng.unimelb.edu.au` and your NeCTAR cloud VM – especially examining what happens when the number of concurrent `clients` increases.

You must add the file `report.txt` to your SVN repository.

Program execution / command line arguments

To run your `sever` program on `digitalis.eng.unimelb.edu.au` (or your NeCTAR cloud VM)

```
prompt: ./server [port_number] [default_secret_code]
```

where `[port_number]` is a valid port number (e.g., 6543)

where `[default_secret_code]` is a valid secret code (e.g., AAAA)

To run your `client` program on `digitalis.eng.unimelb.edu.au`

```
prompt: ./client localhost [port_number]
```

where `localhost` corresponds to 'this computer'

and `[port_number]` is the valid port number (e.g., 6543).

To run your `client` program on a different host machine

```
prompt: ./client [host_name/IP_address] [port_number]
```

where `[host_name/IP_address]` corresponds to host of your `sever`

and `[port_number]` is the valid port number (e.g., 6543).

Note: A new log file `log.txt` file is created each time the `sever` program is started.

Submission details

Please include your *name* and *login_id* in a comment at the top of each file.

Our plan is to directly harvest your submissions on the due date from your SVN repository.

`https://svn.eng.unimelb.edu.au/comp30023-S1-16/username/project2`

You must submit program file(s), including a `Makefile`. Make sure that your makefile, header files and source files are added/committed. Do not add/commit object files or executables. Anything you want to mention about your submission, write a text file called README.

If you do not use your SVN repository for the project you will NOT have a submission and will be awarded zero marks.

It should be possible to "checkout" the SVN repository, then type `make server` to produce the executable `server` and `make client` to produce the executable `client`.

Late submissions will incur a deduction of 2 mark per day (or part thereof).

If you submit late, you MUST email the lecturer, Michael Kirley <`mkirley@unimelb.edu.au`>. Failure to do will result in our request to sysadmin for a copy of your repository to be denied.

Extension policy: If you believe you have a valid reason to require an extension you must contact the lecturer, Michael Kirley <`mkirley@unimelb.edu.au`> at the earliest opportunity, which in most instances should be well before the submission deadline.

- It is university policy that projects/assignment work cannot have a due date that falls during 'Swotvac' (the week before exams starts).
- Requests for extensions are not automatic and are considered on a case by case basis. You will be required to supply supporting evidence such as a medical certificate.

Plagiarism policy: You are reminded that all submitted project work in this subject is to be your own individual work. Automated similarity checking software will be used to compare submissions. It is University policy that cheating by students in any form is not permitted, and that work submitted for assessment purposes must be the independent work of the student concerned.

Using SVN is an important step in the verification of authorship.

Assessment

Code that does not **compile** and **run** on `digitalis.eng.unimelb.edu.au` will be awarded **zero** marks. Your submission will be tested and marked with the following criteria:

- **Client-Server interactions (3 marks)**
  - `server` and `client` programs run on `digitalis.eng.unimelb.edu.au`
  - `client` program displays appropriate messages (and the status of the game is displayed to *stdout*) at each stage of the game
  - it is possible to use the `server` and `client` programs to play a complete game correctly
- **Concurrency (3 marks)**
  - `server` program is able to process a large number of concurrent games (from a variety of IP addresses; up to 20 `client`s) and record interactions in the log file

    Note: it it important that your `client` program only accepts the correct input values (a string of four characters e.g., `ABCD`) representing the guess at each stage of the game – this will allow for automated testing of your submission
- **Log file (3 marks)**
  - correctly documents interactions between the `server` and multiple `client`s using the format specified in the 'Playing the game' section.
  - the 'performance' and resource usage data is written to the log file after the key combination `Ctr+C` is used to 'terminate' the `server` program
- **NeCTAR cloud deployment (2 marks)**
  - **upload to your SVN repository a text file `ip.txt` that contains the IP number for your VM and port number – without these details, we cannot test your submission**.
  - `server` running on your VM functions correctly when tested

    Note: it is your responsibility to make sure that your instance of the VM is running on the NeCTAR cloud
- **Quality of code (2 marks)**
  - elegant code; detailed documentation where necessary
  - coding standards followed (and consistent), Makefile works, use of header files, separate source files (where appropriate)
- **Report (2 marks)**
  - `report.txt` **added to your SVN repository**
  - the report describes additional statistics – it includes more than simply counting `client` 'connects' and games won
  - comparison data included; discussion valid.